gafro: Geometric Algebra for Robotics

Tobias Löw*†, Philip Abbet* and Sylvain Calinon*†
*Idiap Research Institute, Martigny, Switzerland
†EPFL, Lausanne, Switzerland

***Abstract*—Geometry is a fundamental part of robotics and there have been various frameworks of representation over the years. Recently, geometric algebra has gained attention for its property of unifying many of those previous ideas into one algebra. While there are already efficient open-source implementations of geometric algebra available, none of them is targeted at robotics applications. We want to address this shortcoming with our library *gafro*. This article presents an overview of the implementation details as well as a tutorial of *gafro*, an efficient C++ library targeting robotics applications using geometric algebra. The library focuses on using conformal geometric algebra. Hence, various geometric primitives are available for computation as well as rigid body transformations. The modeling of robotic systems is also an important aspect of the library. It implements various algorithms for calculating the kinematics and dynamics of such systems as well as objectives for optimization problems. The software stack is completed by Python bindings in *pygafro* and a ROS interface in *gafro_ros*.**

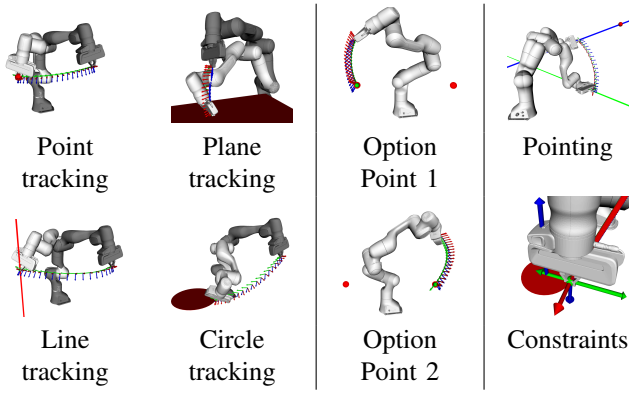


Fig. 1: RViz visualizations of the Franka Emika robot reaching various geometric primitives. These visualizations were created using the tools from *gafro_ros*.

I. INTRODUCTION

Robotics contains very complex problems due to the large variety of different platforms, environments, tasks and interactions. Traditionally, to tackle these complexities in a formal manner, research in robotics introduces different abstraction layers that aim to enable precise reasoning on a semantic level. Different mathematical representations of these abstractions may lead to expensive conversions or even to violations of assumptions in the form of singularities or discontinuities. In robotics, often the problems are fundamentally problems of geometry, hence it is very beneficial to choose representations that intuitively allow to incorporate the geometry of the problem, since the problem and its geometric structure are deeply interconnected.

Geometric algebra (GA) can be considered a high-level mathematical language for geometric reasoning. As such it is very well suited for general problems in robotics. GA unifies the geometric understanding of screw theory, the thoroughness of Lie Algebra and the simplicity of spatial algebra. The representational advantage of geometric algebra is that its elements directly represent geometric objects that can be manipulated by algebraic operations. Complex relations and algorithms can be formulated in a simplified and coordinate-independent way. Furthermore, the existence of different geometric primitives in the same algebra allows for the uniform definition of distance functions, which we will later show in the examples for solving inverse kinematics problems. Dual quaternion algebra is closely related to GA due to their common roots in Clifford algebra. GA is, however, more general and can be defined over any dimension.

The story of geometric algebra in engineering is the story of an algebraic framework that greatly simplifies well-known equations, the most popular example being the Maxwell equations, which reduce to a single equation in geometric algebra. In [1] a survey is presented detailing this story of the development of GA in engineering applications and how it is a powerful geometric language that connects and unifies many mathematical concepts. Another recent survey showing how the applications of GA include physics, electrical engineering, computer graphics to quantum computing, neural networks, signal processing and robotics can be found [2].

Although geometric algebra has great potential for modeling, learning and control in robotics, it has not been widely adopted in robotics research. One reason for this is the lack of easy-to-use libraries for robotics applications, while at the same time tools based on matrix algebra are very mature and readily available. We aim to change that by providing a ready-to-use geometric algebra library for robotics that can be used with the most popular programming frameworks, namely C++, Python and ROS. In our *gafro* library we provide an implementation of the GA variant of algorithms to compute the kinematics and dynamics of robots. These algorithms are well studied and have been implemented in

This work was supported by the State Secretariat for Education, Research and Innovation in Switzerland for participation in the European Commission's Horizon Europe Program through the INTELLIMAN project (https://intelliman-project.eu/, HORIZON-CL4-Digital-Emerging Grant 101070136) and the SESTOSENSO project (http://sestosenso.eu/, HORIZON-CL4-Digital-Emerging Grant 101070310).

various software frameworks. It is important to point out that geometric algebra can be used to compute these important quantities that are classically computed using matrix algebra, while also offering a richer toolset, i.e. it also includes tools for geometric reasoning that matrix algebra does not have. These tools include the construction of geometric primitives that are covariant under motion. The primitives can directly be used for projections, reflections and intersections. All operations do not depend on the choice of an origin, i.e. they are coordinate-free definitions. We demonstrate in this article how these geometric primitives can be used in robotics to facilitate and unify the formulation of control laws and optimization problems.

In this article we want to explain the implementation details of our geometric algebra library *gafro* and show how to use it for common robotics problems such as inverse kinematics and optimal control. We compare *gafro* with existing libraries for geometric algebra and robot modeling. These libraries can be seen in Table I. Our aim is to make geometric algebra more accessible for robotics research by providing this ready-to-use library. This should result in a wider adoption and facilitate research on using this powerful framework for robotics.

This article is organized as follows: in Section II we give a brief introduction to geometric algebra, in Section III we explain the implementation details of the algebra, in Section IV we compare *gafro* to other GA and robot modeling libraries and finally in Section V we demonstrate various applications and give a tutorial on how to use the library. The documentation and the links to all repositories can be found on our website `https://geometric-algebra.tobiloew.ch/gafro`.

TABLE I: Comparison of different libraries.

(a) Overview of other geometric algebra libraries.

Garamon	[3]	a generator of C++ libraries dedicated to geometric algebra
GATL	[4]	C++ library for Euclidean, homogeneous/projective, Minkowski/spacetime, conformal, and arbitrary geometric algebras using template meta-programming
Versor	[5]	(fast) generic C++ library for geometric algebras
GAL	[6]	C++17 expression compiler and engine for computing with geometric algebra
Gaigen	[7]	code generator for geometric algebra
Gaalet	[8]	C++ library for evaluation of geometric algebra expressions offering comfortable implementation and reasonable speed by using expression templates and meta-programming techniques
Gaalop	[9]	software to optimize geometric algebra files
TbGAL	[10]	C++/Python library for Euclidean, homogeneous/projective, Minkowski/spacetime, conformal, and arbitrary geometric algebras representing blades (and versors) in their decomposed state to scale to scale high dimensions

(b) Overview of other libraries for robot modeling.

DQ Robotics	[11]	library for robot modeling and control based on dual quaternion algebra
Pinocchio	[12]	state-of-the-art rigid body algorithms for poly-articulated systems
Raisim	[13]	multi-body physics engine for robotics and AI
KDL	[14]	application independent framework for modeling and computation of kinematic chains
Mujoco	[15]	physics engine for model-based optimization
RBDL	[16]	highly efficient code for both forward and inverse dynamics for kinematic chains and branched models

II. GEOMETRIC ALGEBRA

In this section we give a brief introduction to the mathematical background of geometric algebra. A more comprehensive introduction can be found in [17] including applications in engineering and [18] presents applications in robotics. We also recommend the website `https://bivector.net`, which contains many useful introductory videos.

Formally, geometric algebra $\mathbb{G}_{p,q,r}$ is defined as an associative algebra over the quadratic space $\mathbb{R}^{p,q,r}$, where p, q, and r are the number of basis vectors that square to 1, -1, and 0, respectively and therefore the dimension is $n = p + q + r$. Its algebraic product is called the geometric product

$$\boldsymbol{ab} = \boldsymbol{a} \cdot \boldsymbol{b} + \boldsymbol{a} \wedge \boldsymbol{b} \tag{1}$$

and is the combination of an $\cdot$ inner and an $\wedge$ outer product. The inner product is related to the metric of algebra, whereas the outer product spans vectors to k-vectors, where k refers to the number of linearly independent basis vectors. These elements form the algebraic basis of a geometric algebra and are called basis blades. There are $2^n = 2^{p+q+r}$ basis blades for a given geometric algebra $\mathbb{G}_{p,q,r}$. A general element in GA is called a multivector and is the linear combination of basis blades. The specific variant that we are using is called conformal geometric algebra (CGA) and is denoted as $\mathbb{G}_{4,1}$. For the introduction of the conformal model in the algebra, CGA uses a change of basis which introduces the two null vectors $\boldsymbol{e}_0$ and $\boldsymbol{e}_\infty$, which can be understood as a point at the origin and one at infinity, respectively. In total there are 32 basis blades of grades 0 to 5 in CGA with the structure that can be seen in Figure 2. This high dimension appears to lead to an increased complexity, in practice; however, these multivectors usually are very sparse, a fact that we exploit in our implementation.

The blades of geometric algebra effectively lead to computations with subspaces of the underlying vector space that can be used to represent geometric primitives directly within the algebra. These primitives in CGA include points, lines, planes, circles and spheres. Their construction utilizes the outer product of points

$$X = P_1 \wedge \ldots \wedge P_n, \tag{2}$$

where two points and the point at infinity form a line, three points a circle and four points a sphere. This outer product construction leads to a nullspace representation, i.e. the set of all points that results in zero under the outer product.

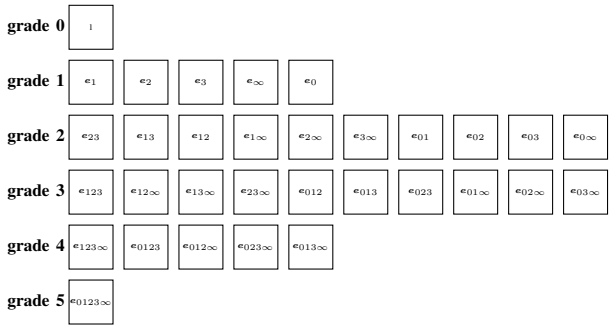


Fig. 2: Structure of conformal geometric algebra with the 32 basis blades, divided into the different grades. Grade 0 and 5 are the scalar and pseudo-scalar, respectively. Grades 1 to 4 are called bi-, tri- and quadvectors.

It is therefore called the outer product nullspace and is defined as the primal representation of the primitives. The dual representation, i.e. the inner product nullspace, can be found via the duality operation, which corresponds to a product with the pseudo-scalar, i.e. the highest grade element of the algebra $\boldsymbol{e}_{0123\infty}$. This construction can expanded to form more complex geometric primitives such as ellipsoids, hyperboloids or general quadric surfaces. We show the specific subspaces that certain primitives of CGA occupy in Figure 3. They can be used for incidence computations, e.g. we can find their intersections by applying product operations, that are known as the meet operator $\vee$

$$Y = X_1 \vee X_2 = (X_1^* \wedge X_2^*)^*, \tag{3}$$

where * denotes the dual operation. This meet operator is singularity-free and geometrically consistent, e.g. the meet of two spheres will result in a circle if the spheres intersect, a point if they only touch each other, or an imaginary circle with a radius related to the distance between the spheres if they are far from each other.

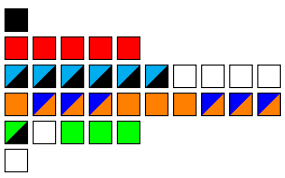


Fig. 3: Non-zero elements of various geometric primitives in their primal representations in conformal geometric algebra. Boxes represent basis blades and colored boxes represent the non-zero blades of the geometric primitive with the matching color. It can be seen that of the 32 basis blades composing multivectors only a sparse number is used for the representations. Note that geometric primitives are single-grade objects, while transformations are mixed-grade.

The geometric primitives can also directly be used for geometric operations such as reflections and projections, which result in rigid body motions. Here, two consecutive reflections on intersecting planes result in a rotation and on parallel planes in a translation. More generally these rigid body transformation are called motors M in geometric algebra. They form a Lie group as the exponential mapping of bivectors, which is the corresponding Lie algebra. In the case of CGA this Lie group corresponds to the conformal group, i.e. the group of angle-preserving transformations, which includes the rigid Euclidean transformations of $SE(3)$ that are most-commonly used in robotics, i.e. rotations and translations. Motors can be applied to any geometric primitive in the algebra via the sandwiching product

$$Y = MX\widetilde{M}, \tag{4}$$

where $\widetilde{M}$ denotes the reverse of a multivector, which can be thought of as being similar to a conjugation of quaternion. The sandwiching product is a structure-preserving product, i.e. the resulting geometric primitive Y will be of the same type as X. This is a property that linear algebra generally does not have automatically and it would need to be explicitly designed and enforced. Furthermore, motors represent a very general concept of rigid body transformations, i.e. as reflections in hyper-planes, which is valid in any dimension.

III. IMPLEMENTATION OF CONFORMAL GEOMETRIC ALGEBRA

In this section we will explain in detail our implementation of Conformal Geometric Algebra (CGA). The aspects that are highlighted are the implementation of a general multivector and the expressions that are acting on it. In this section we explain the programming interfaces that *gafro* offers. The main library is written in C++ for which we provide Python bindings called *pygafro* as well as the ROS package *gafro_ros*. All mentioned repositories can be found at `https://gitlab.com/gafro`. An overview of the software stack can be found in Table II.

TABLE II: Overview of the *gafro* software stack.

gafro	core C++20 library
pygafro	Python bindings
gafro_ros	interface to ROS and URDF
gafro_benchmarks	robot kinematics/dynamics benchmarks
gafro_examples	various code examples
gafro_robot_descriptions	classes defining different robot models

A. Design Goals and Implementation Details

We had several design goals in mind when designing the library and additionally wanted to cover several points that were proposed in [19] as a wishlist for geometric algebra implementations. Since it is targeted at robotics applications including robot learning, control and optimization, we wanted to ensure fast and efficient computation. To this end, the core implementation of *gafro* is done in C++20 and relies heavily on templates, which also serve the additional purpose of alleviating the effect of numerical imprecision that is known to

occur in geometric algebra implementations by only evaluating elements of the resulting multivectors of expressions that are known to be non-zero. We have designed the library in an object-oriented way, so the classes also reflect the mathematical inheritance relationships. Furthermore all classes, i.e. all specialized multivectors, are instantiated as different types, which allows them to be distinguished at compile time for type-safety and to have persistent storage. These specialized classes also enable the computation with partial multivectors, i.e. the library exploits the fact that the most commonly-used multivectors are sparse and only use certain subspaces of the algebra. Mathematical operations are implemented as expression templates, which lets us determine the type of resulting multivectors at compile time. The implementation via expression templates allows the allocation of memory only if the expression is evaluated and also enables partial evaluations. We handle the type explosion of binary operators by automatically evaluating partial expressions when the full expressions get too complex.

In terms of using the library, we wanted to provide an accessible interface and ensure seamless integration with existing software. Geometric algebra is currently not well known in robotics, hence it can be daunting having to simultaneously learn about the algebra and the software implementation. Therefore, *gafro* provides the computation of the most important quantities for robot kinematics and dynamics with an interface that is close to similar robotics libraries. By basing our implementation on the *Eigen*[1] library, these quantities can be returned in the familiar vector/matrix format. This essentially makes *gafro* a drop-in replacement for other kinematics and dynamics libraries, without the need to know about all the details of geometric algebra at first. Afterwards, however, the geometric modeling of primitives, the singularity-free incidence computations, the uniform distance computation, the connections to differential geometry and the conformal group as well as the general structure of the algebra offer distinct advantages compared to other libraries.

B. General Multivector

The core element of computation in geometric algebra is the multivector. Hence, it is very important to think about the design choices when implementing its structure, as this will determine the memory usage and computational performance. The general structure of a multivector in CGA can be seen in Figure 2. It is composed of 32 basis blades, divided into grades zero to five. A general multivector would therefore be quite heavy in terms of memory and computation. The important structural aspect of CGA that facilitates the design process here is the sparsity of its representations and the fact that the structure of multivector expressions is known at compile time. Both of these properties mean that we can implement the data vector of a multivector by only storing its known non-zero elements. This is achieved by using a template that takes list of blade indices as input:

```
template <class T, int... blades>
class Multivector
{
    public:
        constexpr static int size = sizeof...(blades);
        {...}
    private:
        Eigen::Matrix<T, size, 1> data_;
};
```

The list of indices is then stored internally as a bitset that facilitates the comparison of the subspaces of two multivectors. A bitset is simply a list of 32 bits that are either 0 or 1, depending on whether the corresponding blade is present in the multivector or not. The memory that is allocated corresponds to the number of blade indices that is given to the template. It uses an `Eigen::Matrix` to store the data, which is exposed via an accessor function called `vector()`. This makes it possible to directly use the parameter vector of any multivector, which is useful for e.g. optimization solvers. The `Multivector` class and all its derived classes (including the expressions) have a method called `get` that is templated on the blade index. This method is fundamental to the design of the library, since all expressions use it for evaluation. Therefore, we add a template constraint using the `requires` keyword of C++20 to ensure that multivector expressions only compile if they contain the requested blade index.

The underlying data type `T` is a template argument, which makes it possible to either use e.g. `float` or `double`, depending on the system architecture. Furthermore, it allows the usage of general purpose automatic differentiation libraries such as *autodiff*[2]. This helps when formulating optimization problems in geometric algebra using *gafro* since it facilitates the coding of complex objective functions and thus accelerates prototyping. Another relevant data type is the `torch::Tensor` class of the *libtorch* library[3], i.e. the C++ distribution of *PyTorch*. This effectively allows parallel computations with geometric algebra, which is important for various methods related to robot learning. While this combination can already be used, we are currently developing a library *gafro_torch* that facilitates the usage.

C. Algebraic Computations using Expression Templates

In order to do algebraic computations, there are several required operations on multivectors, which are implemented as expression templates. There are two types of expressions, unary and binary expressions, which are listed in Tables III and IV, respectively. We explained their mathematical meaning in Section II and thus focus here on their implementations.

TABLE III: Unary expressions that are implemented as member functions of the `Multivector` class.

member function	mathematical symbol
`reverse`	$\widetilde{X}$
`inverse`	X^{-1}
`dual`	X^*

[1]https://eigen.tuxfamily.org

[2]https://autodiff.github.io/

[3]https://pytorch.org/cppdocs/

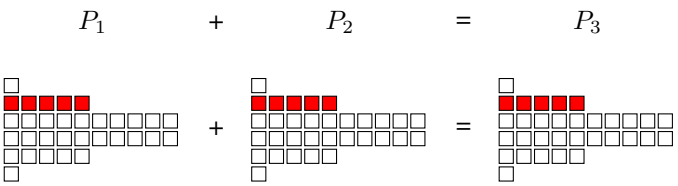
TABLE IV: Binary expressions. The term operator here refers to the programming operators that are implemented for multivectors and symbol means the mathematical symbol that is used in the equations. Note that usually the geometric product is written in equations without a symbol, e.g. AB.

	operator	symbol
addition	+	$+$
subtraction	-	$-$
outer product	^	$\wedge$
inner product	|	$\cdot$
geometric product	*	

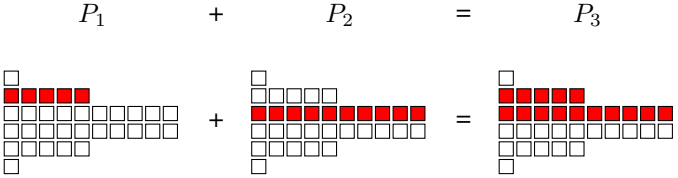
The challenge in the implementation is the fact that the resulting multivectors only rarely have the same blades as the input operands. Given the structure of the algebra, however, the expression templates can determine the result type of the expression at compile time. Note that the expressions are evaluated in a lazy fashion, which means that the blades are evaluated on demand. This makes it possible to for example only evaluate a single blade of the resulting multivector.

```
template <class Derived, class Result>
class Expression
{
    public:
        template <int blade>
        requires(Result::has(blade))
        typename Result::Vtype get() const
        {
            return static_cast<const Derived &>(*this).
                template get<blade>();
        }
};
```

A first example of this can be seen in the `Sum` expression in Figure 4. The corresponding type evaluation class constructs the type of the resulting multivector at compile time. In the case of addition this amounts to a simple bitwise OR operation comparing the bitsets of the input multivectors.



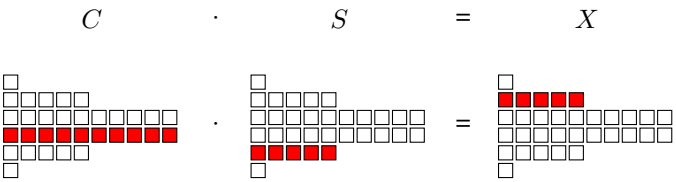
(a) The addition of two multivectors with the same blades results in another multivector with the same blades.



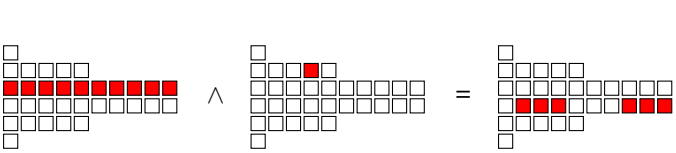
(b) The addition of two multivectors with the different blades results in a multivector with the blades of both input multivectors.

Fig. 4: Addition operation.

The inner and outer products work essentially in the same way and thus the corresponding expression both inherit from a base `Product` class, i.e. `InnerProduct` and `OuterProduct`. The `Product` class takes a class structure implementing the corresponding Cayley table as template argument. This Cayley table defines the resulting blades of a blade by blade product under the inner and outer product, respectively. Thus, in the case of CGA, it defines 1024 operations. In order to determine the type of the resulting multivector, we employ fold expressions that allow us to iterate over the blades of both input multivectors at compile time. In this loop, we obtain the resulting blade per pair of blades using the respective Cayley table and then assemble them into the resulting multivector again using OR operations. Figure 5 shows an example for each the inner and the outer product.



(a) The inner product is a grade-lowering operation, i.e. the resulting multivector will be of lower grade than the inputs. The example shows that the inner product of a circle C with a sphere S results in a point P.



(b) The outer product is a grade-raising operation, i.e. the resulting multivector will be of higher grade than the inputs. The example shows that the outer product of a point pair PP and $\boldsymbol{e}_\infty$ results in a line L

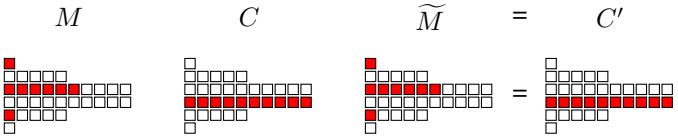
Fig. 5: The resulting multivector of the inner and outer product operations has a different grade than the inputs.

The geometric product class `GeometricProduct` also inherits from the base `Product` class and comes with its own Cayley table. So implementation-wise it is the same as the inner and outer products. The main difference is that two blades can result from a blade product, which causes the resulting multivector to potentially have both a lower and a higher grade than the inputs, as can be seen in Figure 6a. Furthermore, we have products that are based on the geometric product such as the sandwich product, which is also treated as a binary expression and shown in Figure 6b.

D. Geometric Primitives

Since we know the subspaces of all the geometric primitives, we chose to implement them in an object-oriented way by inheriting from the base `Multivector` class. Hence, the available classes are `Vector`, `DirectionVector`, `TangentVector`, `Point`, `PointPair`, `Line`, `Circle`, `Plane` and `Sphere`. Their corresponding subspaces within the geometric algebra can be seen in Figure 3. Having the

(a) Using the geometric product results in both blades of lower and higher grade.



(b) The sandwich product is a grade-preserving operation. Numerical issues might lead to residuals in other blades, which we avoid by simply not evaluating them in the expressions. This is a schematic representation of Equation (4), where a motor transforms a circle.

Fig. 6: The geometric product is a combination of the inner and outer product.

geometric primitives as explicit classes allows the implementation of commonly-used equations as members functions, which facilitates the usage. For example the constructors of the geometric primitive classes implement the various ways they can be defined. The explicit classes are meant to facilitate the use and construction, but of course, using computation with base multivectors is also possible. This preserves the property of covariant computation within the algebra.

E. Rigid Body Transformations

The rigid body transformations that are currently available are implemented in the classes `Rotor`, `Translator`, `Motor` and `Dilator`. They all inherit from the base class `Versor`. Since all three classes are exponential mappings of bivectors they are accompanied by the expressions `Logarithm` and `Exponential`, respectively. The main method of the rigid body transformations is $\widetilde{\texttt{apply}}$, which implements the sandwich product $X' = MX\widetilde{M}$ and ensures type safety. While X can technically be any multivector, the intended usage is with the geometric primitives that were presented in Section III-D. Hence, in this context, type safety means that X' stays the same geometric primitive as X, e.g. a `Point` stays a `Point`. This ensures that the expression only evaluates blades that are part of the geometric primitive, which not only reduces the number of floating-point operations, but also deals with numerical imprecision in the computation that is known to occur in geometric algebra implementations.

F. Robot Modeling

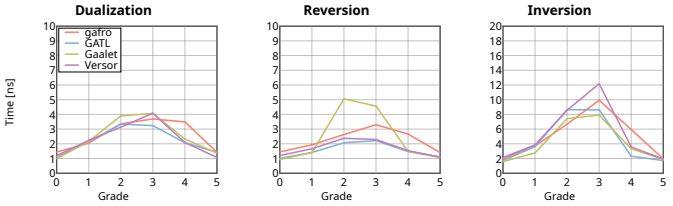
The previous sections introduced the features related to the underlying geometric algebra implementation of *gafro*. This section will now introduce the higher level features of the library related to robot modeling, which distinguish it from other geometric algebra libraries. The main aspects of robot modeling are the computation of the kinematics and dynamics of robotic systems, which is implemented in the base class called `System`. It contains member functions that compute the forward kinematics and forward/inverse dynamics using recursive algorithms. We also provide classes to model optimization problems for robotic systems, such as inverse kinematics. We will present more on this in Section V with concrete examples.

IV. COMPARISON TO OTHER LIBRARIES

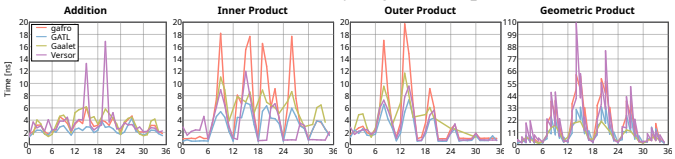
In this section, we compare *gafro* to other geometric algebra and robot kinematics/dynamics libraries. We first provide quantitative benchmark results and then give qualitative comparisons of what we believe to be advantages of *gafro* over other libraries.

A. Algebraic Operations Benchmarks

In order to compare the performance of our *gafro* library to other geometric algebra libraries we forked the *ga_benchmark*[4] repository in order to integrate *gafro*. Our fork can be found at `https://github.com/loewt/ga-benchmark`.



(a) Benchmarks of unary algebraic operations.



(b) Benchmarks of binary algebraic operations.

Fig. 7: Benchmarks of different geometric algebra libraries. All operations are computed using conformal geometric algebra. Some entries for *Gaalet* are missing due to segmentation faults during the execution.

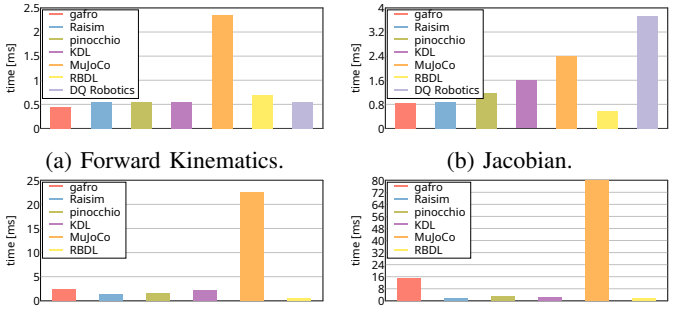
We omitted *TbGAL* and *Garamon* from the plots of the benchmark results, since they are by far the slowest libraries. The benchmarks show that *gafro* can compete in terms of performance with *GATL* and *Versor*, which were previously reported to be the fastest GA libraries.

B. Robotics Algorithms Benchmarks

Since this library implements robot kinematics and dynamics algorithms, we are comparing and benchmarking *gafro* against several libraries that are commonly used in robotics applications. The current benchmarking results on our system can be found in Figure 8. As can be seen, *gafro* is very competitive when it comes to the computation of the kinematics of a robotic system. These advantages come from the fact that motors in geometric algebra are a more compact

[4]https://github.com/ga-developers/ga-benchmark

representation and require fewer arithmetic operations than transformation matrices. The computation of the dynamics, however, especially the forward dynamics, is still slower at this point. This is because at this stage we were prioritizing the research aspect of the algorithms, since they needed to first be derived in CGA. This means that the forward dynamics present a novelty not only in the implementation, but also in the mathematical derivation. This lead to a naive implementation, which includes unnecessary copy operations that affect the performance negatively. The issue will be addressed and fixed in a future release of *gafro*, which should make the computation of the dynamics also competitive compared to the established libraries.



(a) Forward Kinematics. (b) Jacobian.

(c) Inverse Dynamics. (d) Forward Dynamics.

Fig. 8: Benchmarks of robotics algorithms. The benchmark was run on an AMD Ryzen 7 4800U CPU. All libraries were compiled using *gcc 13.1.1* with the compiler flags `-O3 -march=native`. The reference system is the Franka Emika Robot.

C. Advantages of gafro

There have been various works that published implementations of geometric algebra. These libraries all have in common that they are meant to be generic geometric algebra implementations focusing on the computational and mathematical aspects of the algebra itself. In contrast to that, our implementation is targeted specifically at robotics applications and thus not only implements the low-level algebraic computations but also features the computation of the kinematics and dynamics of serial manipulators as well as generic cost functions for modeling optimization problems. We will explain these cost functions in detail in Section V-C. Here, we want to point out that these cost functions simultaneously present an advantage over other geometric algebra libraries and over other robot modeling libraries, since neither of them target the geometric modeling robotics problems. *gafro* can therefore be seen as bridging the gap between these libraries.

We further believe that the programming interface of *gafro* is a lot more approachable and easy to use than other geometric algebra libraries. One reason for this is that we provide explicit classes for the geometric primitives and by making use of the C++ constructors, they can be created and used without having to explicitly use the algebraic operations. These classes also allow us to implement commonly used operations of multivectors directly as member functions, such as the sandwich product of motors to transform geometric primitives.

V. APPLICATIONS AND TUTORIAL

In this section we provide some example applications of how the library can be used. For that purpose, we provide an accompanying repository *gafro_examples* that contains coding examples. Note that in the text we are always referencing the C++ files, but the same examples can also be found in Python in the corresponding folder. These examples use the same naming scheme.

A. Geometric Algebra

Since many potential users of *gafro* are likely to be unfamiliar with the concept of geometric algebra we are providing some examples on how to do computations using this algebra. In a first example we are showing how to create different general multivectors and use them for algebraic computations. For this purpose we demonstrate how to compute the intersection of a sphere S and a plane E, which in this case results in a circle C. The construction of a plane requires three points and the one at infinity $\boldsymbol{e}_\infty$ and the construction of a sphere requires four points. Hence, we first define seven P_i using their Euclidean coordinates.

```
gafro::Point<double> p1(x, y, z);
```

From these points we the calculate the plane and the sphere, where `Ei` corresponds to $\boldsymbol{e}_\infty$.

```
gafro::Plane<double> plane = p1 ^ p2 ^ p3 ^ gafro::Ei<
    double>(1.0);
gafro::Sphere<double> sphere = p4 ^ p5 ^ p6 ^ p7;
```

Note that here we choose to construct the plane and the sphere by the outer product of points, according to their mathematical definition, which derives from Equation (2). They could, however, be equivalently created by passing the same points to the respective constructors, which we show in the coding examples in the online repository.

The circle primitive that is found from the intersection of a plane and a sphere, which is expressed mathematically as the meet operator in Equation (3). This Equation can directly be translated to code to find the circle.

```
gafro::Circle<double> circle = (plane.dual() ^ sphere.dual
    ()).dual();
```

Note that this code will compile and can be executed successfully without runtime errors whether the plane and the sphere intersect or not. This geometric relationship can be determined from the resulting circle by inspecting the squared norm, which will be positive or negative, depending on the incidence relationship.

B. Robot Differential Kinematics

One of the targeted use cases of the *gafro* library is the modeling of robotic systems. In this section, we will show how to do that in practice by explaining the example of a differential kinematics controller that tracks line in task space using an arbitrary reference line at the robot end-effector.

We use in this example the class `FrankaEmikaRobot` and assume we have instantiated it as `panda`.

First, the forward kinematics, i.e. the pose of the end-effector of a kinematic chain given a certain joint configuration, are represented by the motors in geometric algebra. For a given joint configuration `q` the end-effector motor is found

```
gafro::Motor<double> ee_motor = panda.getEEMotor(q);
```

We implement the differential kinematics controller w.r.t to the robot end-effector frame. Hence we use the end-effector motor for transforming an object of type `gafro::Line` called `target_line` to this frame. We skip the creation of this line here, but note that it is similar to the creation of geometric primitives in the previous section. The transformation of the line using `ee_motor` is implemented as follows.

```
gafro::Line<double> transformed_line = ee_motor.reverse() *
    target_line * ee_motor;
```

This effectively corresponds to the equation $L' = \widetilde{M}LM$, as opposed to $L' = ML\widetilde{M}$, which was shown previously in Equation (4). The difference is that here we use the inverse transform from the base frame to the end-effector frame.

Next, we find the twist, which moves the reference line to the transformed target line. This twist is found as the logarithmic mapping of the motor that transforms one line to the other. Mathematically this can be expressed as

$$\mathcal{V} = \log\left(\frac{1}{c}\left(1 + L_r L_t\right)\right), \tag{5}$$

where $\mathcal{V}$ is the resulting twist and c is a normalization constant. L_r and L_t are the reference and target line, respectively. We have implemented this as member function of the `Line` class, such that it can be called directly as

```
gafro::Twist<double> twist = transformed_line.getMotor(
    reference_line).log();
```

The last step is the computation of the joint velocities $\dot{\boldsymbol{q}}$ from the twist $\mathcal{V}$. This is achieved using the inverse of the end-effector frame Jacobian, which can be obtained by the member function `getEEFrameJacobian` of the `panda` robot. This function returns a *gafro* specific object, which can be transformed to an `Eigen::Matrix` using the `embed` method. The control law according to the equation $\dot{\boldsymbol{q}} = \boldsymbol{J}^{-1}\mathcal{V}$ can therefore be implemented as

```
Eigen::Vector<double,7> qdot = inverse(panda.
    getEEFrameJacobian(q).embed()) * twist.vector();
```

Here we use the `inverse` function as shorthand for the pseudo-inverse of a 6×7 matrix.

The lines in this example can be chosen arbitrarily, which is a very appealing property, since it has two important consequences. First, the reference line at the end-effector constrains two axes of rotation, while allowing a rotation around the line. This line does not need to coincide with the axes of the end-effector frame. The axes are not even required to be known explicitly, the line is sufficient. This essentially avoids having to deal with coordinate frames when encoding the target. And second, the two lines are invariant under translations along them, i.e. moving a line in the direction its pointing does not change the line. In practice, this effectively means that the control law is completely compliant to disturbances along the superposed lines. These two properties are very hard to achieve using classical methods and require many coordinate frame changes and non-trivial precision matrices. This example shows that the definition of a control law using geometric primitives can be done entirely geometrically and resulting equations are very simple since they are also algebraic objects.

C. Optimization Problems with Geometric Primitives

Many problems in important domains of robotics, such as learning and control, can be cast as optimization problems. Hence, in this section we are providing an example on how *gafro* can be used for the uniform modeling of optimization problems using geometric algebra. Here, we cast the optimization problem as an inverse kinematics problem for simplicity, so we are optimizing for the joint angle configuration in which the end-effector reaches a certain geometric primitive and we show how GA extends the cost function to be uniformly applicable across the different geometric primitives. The optimization problem can be formulated as follows

$$\boldsymbol{q}^* = \min_{\boldsymbol{q}} \frac{1}{2}\left\|E(\boldsymbol{q})\right\|^2, \tag{6}$$

where $\boldsymbol{q}$ is the joint angle configuration and $E(\boldsymbol{q})$ is a multivector-valued residual. In *gafro* this formulation is implemented in the generic template class `SingleManipulatorTarget` and Equation (6) can be evaluated using the method `getValue`. The below code snippet of this shows that it has the template arguments `Tool` and `Target`, which are meant to be different geometric primitives.

```
template <class T, int dof, template <class Type> class
    Tool, template <class Type> class Target>
class SingleManipulatorTarget{...};
```

`Tool` is a geometric primitive at the end-effector of the robot arm, e.g. a point, and `Target` is a desired geometric primitive that should be reached by the end-effector, e.g. a line or a circle. The problem of reaching can be expressed as minimizing a distance measure between the two primitives. Mathematically, this distance measure can be expressed as a residual multivector stemming from the outer product, i.e.

$$E(\boldsymbol{q}) = X_d \wedge M(\boldsymbol{q}) X \widetilde{M}(\boldsymbol{q}), \tag{7}$$

where X corresponds to the `Tool` and X_d to the `Target`, the motor $M(\boldsymbol{q})$ is the end-effector motor at the current joint configuration $\boldsymbol{q}$, which transforms any geometric primitive to the end-effector, expressed w.r.t. the base frame. By definition, this outer product results in zero, if `Tool` has reached the `Target`. Its norm therefore corresponds to a distance measure that we want to minimize here. In the implementation this residual multivector from Equation (7) is obtained by calling the function `getResidual`, which can be seen in the code snippet below.

```
Eigen::Matrix<T, Result::size, 1> getResidual(const VectorX
      &q) const
{
    return Result(target_ ^ arm_.getEEMotor(q).apply(tool_)
        ).vector();
}
```

The Jacobian of Equation (7) w.r.t. the joint configuration vector $\boldsymbol{q}$ is found by applying the chain rule to the geometric product of the motor $M(\boldsymbol{q})$, i.e.

$$\boldsymbol{\mathcal{J}}^E(\boldsymbol{q}) = X_d \wedge \left(\boldsymbol{\mathcal{J}}^A(\boldsymbol{q}) X \widetilde{M}(\boldsymbol{q}) + M(\boldsymbol{q}) X \widetilde{\boldsymbol{\mathcal{J}}^A}(\boldsymbol{q})\right), \quad (8)$$

where $\boldsymbol{\mathcal{J}}^A(\boldsymbol{q})$ is the analytic Jacobian of the kinematic. The following code snippet shows the implementation of Equation (8). Both implementations closely follow the mathematical formulation.

```
Eigen::Matrix<T, Result::size, dof> getJacobian(const
     VectorX &x) const
{
    Motor<T> motor = arm_.getEEMotor(x);
    MultivectorMatrix<Motor<T>, 1, dof> jacobian_ee = arm_.
         getEEAnalyticJacobian(x);

    Eigen::Matrix<T, Result::size, dof> jacobian;

    for (unsigned i = 0; i < dof; ++i)
    {
        jacobian.col(i) = Result(target_ ^ (jacobian_ee[i]
            * tool_ * motor.reverse() + motor * tool_ *
            jacobian_ee[i].reverse())).vector();
    }

    return jacobian;
}
```

Note that both `getResidual` and `getJacboian` return a matrix of the *Eigen* library where the size is determined based on the combination of geometric primitives. More specifically the size can vary depending on the primitives, but due to the structure of the algebra and its implementation using expression templates, the size is determined at compile time. In practice, the actual sizes of the residual and Jacobian can be neglected, since for solving an optimization problem, we are actually interested in the gradient vector $\boldsymbol{g} \in \mathbb{R}^{N\times 1}$ and Hessian matrix $\boldsymbol{H} \in \mathbb{R}^{N\times N}$ of Equation (6) and their size is only determined by the number of degrees of freedom N of the robot and is therefore agnostic to the choice of geometric primitives.

Given the residual and the Jacobian, the optimization problem can easily be solved using for example a Gauss-Newton type algorithm. Both of these quantities can be accessed from the class via the method `getGradientAndHessian` which returns them in the form of matrices from the *Eigen* library. This choice fulfills one of the design goals of the library, i.e. the seamless integration with existing optimization solvers. Hence, it is possible to use these geometric algebra computations in existing pipelines, without having to fundamentally rewrite existing software to accommodate the geometric algebra, which keeps the integration effort low. This example is also applicable across a wider range of applications, in previous work, we have shown the application of CGA to modeling manipulation tasks in an optimal control framework for model predictive control [20], which can of course be achieved using the same cost function.

We give several examples of this inverse kinematics problem in *gafro_examples*. The files are following the naming scheme *inverse_kinematics_PRIMITIVE1_PRIMITIVE2.cpp*. We visualize the results of optimization problems using various geometric primitives in Figure 1. We want to point out, that the implementations only differ in the instantiation of the `SingleManipulatorTarget` template class, which shows the ability of geometric algebra to unify formulations and simplify their implementations.

VI. CONCLUSION

In this article we presented the implementation details as well as some examples for our software stack around *gafro*, which is a C++ library that implements conformal geometric algebra for robotics. The software stack also includes Python bindings in *pygafro* as well as a ROS package in *gafro_ros*. Tutorial material and toy examples can be found in *gafro_examples*.

While showing comparable performance for the robot modeling, geometric algebra also offers an easy and intuitive way to model various geometric relationships as shown by examples of intersecting geometric primitives, differential kinematics using line objects and optimization based inverse kinematics with different geometric primitives. The motors are a more general concept of transformations that can be directly applied to all geometric primitives within the algebra, alleviating the the need to compute special adjoint operations. Combined with the fact that motors are a more compact representation of rigid body transformations that requires less operations, geometric algebra offers a very rich and appealing mathematical framework for robotics, without losing any of the existing tools that are offered by standard matrix algebra.

Our library *gafro* provides the standard algorithms for robot modeling and the computation of the kinematics and dynamics. It then augments them with concepts that are exclusive to geometric algebra, such as direct representations of geometric primitives and operations on them which are then used for the implementation of general optimization problems. In fact, by the design of the library, which exposes the parameter vectors using `Eigen`, these standard libraries could directly be replaced by *gafro* without having to use geometric algebra directly. Providing this library that makes geometric algebra easily accessible for robotics research should allow for a wider adoption and facilitate research on using this powerful framework for robotics.

REFERENCES

[1] E. Bayro-Corrochano, "A Survey on Quaternion Algebra and Geometric Algebra Applications in Engineering and Computer Science 1995–2020," *IEEE Access*, vol. 9, pp. 104 326–104 355, 2021. DOI: 10.1109/ACCESS.2021.3097756.

[2] E. Hitzer, M. Kamarianakis, G. Papagiannakis, and P. Vašík, "Survey of new applications of geometric algebra," *Mathematical Methods in the Applied Sciences*, vol. n/a, no. n/a, DOI: 10.1002/mma.9575.

[3] S. Breuils, V. Nozick, and L. Fuchs, "Garamon: A Geometric Algebra Library Generator," *Adv. Appl. Clifford Algebras*, vol. 29, no. 4, p. 69, Jul. 22, 2019. DOI: 10.1007/s00006-019-0987-7.

[4] L. A. F. Fernandes, "Exploring Lazy Evaluation and Compile-Time Simplifications for Efficient Geometric Algebra Computations," in *Systems, Patterns and Data Engineering with Geometric Calculi*, S. Xambó-Descamps, Ed., vol. 13, Cham: Springer International Publishing, 2021, pp. 111–131. DOI: 10.1007/978-3-030-74486-1_6.

[5] P. Colapinto, "Versor: Spatial computing with conformal geometric algebra," University of California at Santa Barbara, 2011.

[6] J. Ong, *GAL*, https://github.com/jeremyong/gal: GitHub, 2019.

[7] D. Fontijne, "Gaigen 2: A geometric algebra implementation generator," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering - GPCE '06*, Portland, Oregon, USA: ACM Press, 2006, p. 141. DOI: 10.1145/1173706.1173728.

[8] F. Seybold and U. Wössner, *Gaalet - a C++ expression template library for implementing geometric algebra*, 2010.

[9] D. Hildenbrand, J. Pitt, and A. Koch, "Gaalop—High performance parallel computing based on conformal geometric algebra," in *Geometric Algebra Computing: In Engineering and Computer Science*, E. Bayro-Corrochano and G. Scheuermann, Eds., London: Springer London, 2010, pp. 477–494. DOI: 10.1007/978-1-84996-108-0_22.

[10] E. V. Sousa and L. A. F. Fernandes, "TbGAL: A Tensor-Based Library for Geometric Algebra," *Adv. Appl. Clifford Algebras*, vol. 30, no. 2, p. 27, Apr. 2020. DOI: 10.1007/s00006-020-1053-1.

[11] B. V. Adorno and M. Marques Marinho, "DQ Robotics: A Library for Robot Modeling and Control," *IEEE Robotics Automation Magazine*, vol. 28, no. 3, pp. 102–116, Sep. 2021. DOI: 10.1109/MRA.2020.2997920.

[12] J. Carpentier, G. Saurel, G. Buondonno, *et al.*, "The Pinocchio C++ library : A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," in *2019 IEEE/SICE International Symposium on System Integration (SII)*, Paris, France: IEEE, Jan. 2019, pp. 614–619. DOI: 10.1109/SII.2019.8700380.

[13] J. Hwangbo, J. Lee, and M. Hutter, "Per-Contact Iteration Method for Solving Contact Dynamics," *IEEE Robot. Autom. Lett.*, vol. 3, no. 2, pp. 895–902, Apr. 2018. DOI: 10.1109/LRA.2018.2792536.

[14] R. Smits, *KDL: Kinematics and Dynamics Library*, http://www.orocos.org/kdl.

[15] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[16] M. L. Felis, "RBDL: An efficient rigid-body dynamics library using recursive algorithms," *Auton Robot*, vol. 41, no. 2, pp. 495–511, Feb. 1, 2017. DOI: 10.1007/s10514-016-9574-0.

[17] C. Perwass, *Geometric Algebra with Applications in Engineering* (Geometry and Computing 4). Berlin: Springer, 2009, 385 pp.

[18] E. Bayro-Corrochano, *Geometric Algebra Applications Vol. II: Robot Modelling and Control*. Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-030-34978-3.

[19] W. Benger and W. Dobler, "Massive Geometric Algebra: Visions for C++ Implementations of Geometric Algebra to Scale into the Big Data Era," *Adv. Appl. Clifford Algebras*, vol. 27, no. 3, pp. 2153–2174, Sep. 2017. DOI: 10.1007/s00006-017-0780-4.

[20] T. Löw and S. Calinon, "Geometric Algebra for Optimal Control With Applications in Manipulation Tasks," *IEEE Transactions on Robotics*, pp. 1–15, 2023. DOI: 10.1109/TRO.2023.3277282.